Linguaggi formali e compilazione

Corso di Laurea in Informatica

A.A. 2015/2016

Linguaggi formali e compilazione

Cenni alle regole di ambiente e generazione del codice intermedio

- Symbol table e regole di ambiente
- Rappresentazioni intermedie
- Dall'AST alla generazione del three-address code
- Traduzione guidata dalla sintassi

Static scoping

- Abbiamo già osservato come le informazioni contenute nella symbol table consentano di controllare il soddisfacimento di alcuni requisiti di correttezza del programma che sarebbe complicato "forzare" nella sintassi
- Esempi riguardano:
  1. il fatto che una variabile sia stata definita prima di venire usata
  2. la concordanza dei tipi
  3. la concordanza sul numero di parametri formali e argomenti di una funzione
- Ora vedremo brevemente come un'accurata realizzazione della symbol table consenta anche di implementare le regole statiche di *ambiente* o di visibilità dei simboli (in inglese *static scoping rule*).

Ambiente

- Lo scope (ambiente) della dichiarazione di una variabile è la porzione di programma in cui tale variabile è visibile (e dunque utilizzabile).
- Lo stesso identificatore può essere definito più volte in un programma, ma ad esso saranno associati ambienti diversi.
- Nei moderni linguaggi di programmazione l'ambiente di una variabile è "quasi sempre" determinabile in modo statico, cioè guardando il testo del programma.
- In particolare, la visibilità è determinata dalla struttura di annidamento dei blocchi di programma.
- In questo caso si parla di *static* (o *lexical*) *scoping*.

Esempio (in Pascal)

```
Program P;
var i: integer;
    c: char;
    x: real;
    function f(x: integer);
    var i: integer;
        procedure z;
        var x: integer;
        begin
         (1)
        end
    begin
     (2)
    end
begin
 (3)
end
```

Static scoping

- L'implementazione della nozione di ambiente può essere realizzata nel compilatore mediante una struttura a stack della symbol table.
- Più precisamente, si utilizzano più tabelle concatenate e la symbol table nel suo insieme è una lista di tabelle.
- Quando incontra l'inizio di un blocco di programma, il compilatore inizializza una tabella e la inserisce in testa alla lista.
- Quando incontra la fine di un blocco rimuove la tabella in testa alla lista.

Static scoping (2)

- L'inserimento di un simbolo avviene sempre nella tabella di testa.
- Il look up della symbol table avviene dapprima nella tabella di testa.
- In caso di search hit, il look up termina, altrimenti procede nella successiva tabella.
- Se il simbolo non viene trovato in alcuna tabella si ha una search miss (con eventuale generazione di un messaggio di errore).

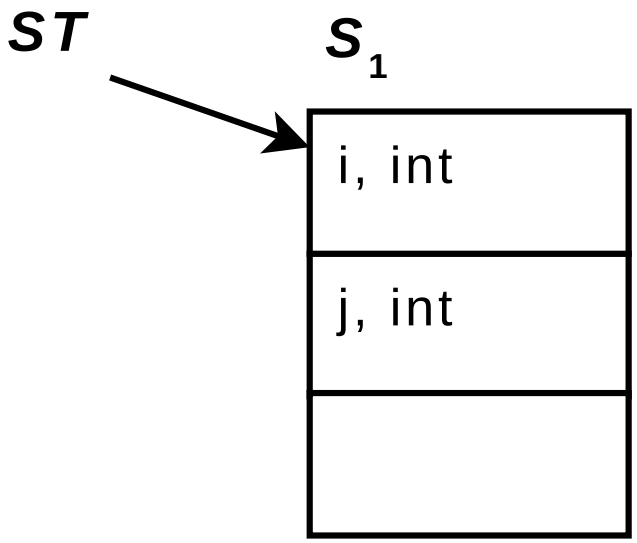
Esempio

- Si consideri, come esempio, il seguente semplice frammento di programma C++:

```
1. { int i,j;
2.   cin » i » j;
3.   { int i; float x;
4.     i=1;
5.     x=2.0*j;
6.     cout « i « ": " « x;
7.   }
8.   cout « i « ": " « j;
9. }
```

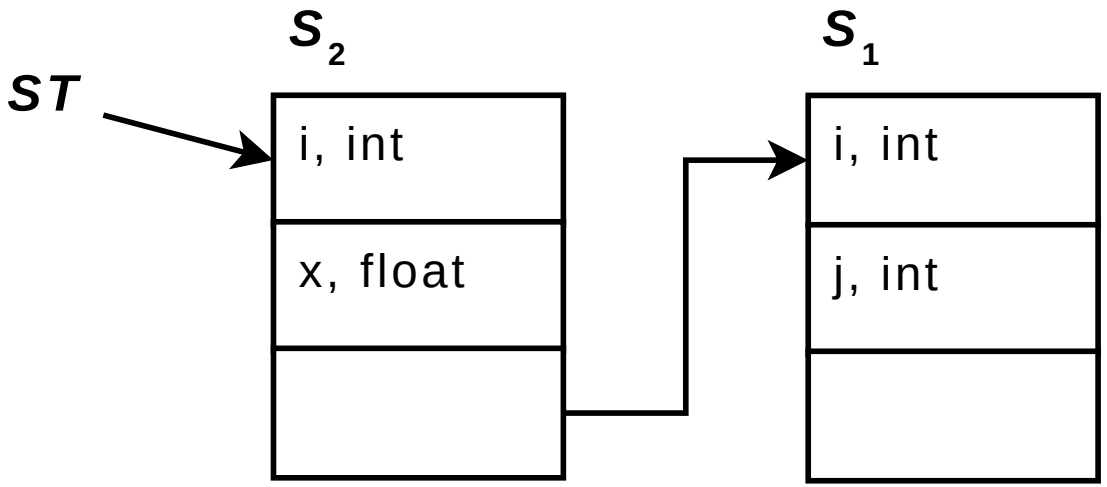
Esempio (continua)

- All'ingresso del blocco più esterno viene creata una tabella S_1 (essenzialmente un dizionario) che diviene la testa della *symbol table*.
- Gli identificatori `i` e `j` di riga 1 vengono inseriti in S_1.



Esempio (continua)

- All'ingresso del blocco interno (riga 3) viene creata (e inserita in testa alla *symbol table*) una seconda tabella, S_2, nella quale vengono poi inseriti gli identificatori `i` e `x` di riga 3 (col loro tipo).



- Il riferimento alla variabile `i` di riga 4 viene risolto con un lookup alla tabella di testa (S_2), che contiene un'entry con chiave `i`.

Esempio (continua)

- Il riferimento alla variabile `j` di riga 5 viene risolto con un lookup alla tabella S_1, dopo aver cercato "inutilmente" in S_2 un'entry con chiave `j`.
- Si noti come la definizione della variabile `i` di riga 3, unitamente al processo di lookup, renda invisibile, nel blocco interno, la variabile `i` definita a riga 1.
- All'uscita del blocco interno, il puntatore alla testa della *symbol table* viene fatto avanzare, con l'effetto di rendere inaccessibile la tabella di testa (S_2).
- Il riferimento alla variabile `i` di riga 8 verrà quindi nuovamente risolto con un lookup a S_1.

Altri usi della symbol table

- Oltre all'implementazione delle regole d'ambiente e a verifiche di correttezza semantica, le informazioni contenute nella symbol table sono fondamentali in sede di generazione del codice
- Ad esempio, la entry per un identificatore contiene:
  - la sequenza di caratteri che ne costituisce il lessema (che può essere usata come chiave per l'accesso alle tabelle);
  - il tipo della variabile;
  - l'indirizzo (relativo) di memoria della variabile.
- Fra gli altri usi, il tipo è importante per stabilire l'occupazione di memoria mentre l'indirizzo serve per generare gli operandi nelle istruzioni macchina.

Le due parti del compilatore

- Il passaggio dal codice sorgente ad un codice macchina efficiente viene tipicamente spezzato in due parti.
  - La prima parte, gestita dal front-end del compilatore, termina con la generazione di un opportuno codice intermedio ed è chiaramente dominata dalle caratteristiche del linguaggio sorgente.
  - La seconda parte, gestita dal back-end, è invece specializzata ad ottenere un codice macchina efficiente in funzione della particolare architettura.
- La suddivisione netta del progetto di un compilatore in front-end e back-end (la prima indipendente dall'architettura e la seconda indipendente dal linguaggio) ha, fra le altre proprietà, un notevole impatto in termini di economicità di progetto.
- Le rappresentazioni intermedie più importanti sono i *syntax tree* e il cosiddetto *three-address code*.

Three address code

- Il codice a tre indirizzi è una rappresentazione intermedia lineare del programma sorgente indipendente da ogni specifica architettura.
- Più precisamente, il three address code è il codice per una architettura astratta di calcolatore.
- Tale modello descrive abbastanza fedelmente la struttura di ogni macchina reale, evitando tuttavia di trattare tutti i complessi dettagli delle moderne achitetture.

Il modello di calcolo per il three address code

- Il calcolatore astratto è in grado di eseguire semplici istruzioni caratterizzate da un *codice operativo* e da *al più* tre indirizzi per gli operandi (da qui il nome).
- Fra le istruzioni disponibili in tale modello di calcolatore, troviamo le operazioni logico/aritmetiche binarie e unarie, le istruzioni di salto (condizionato o incondizionato), il semplice assegnamento, la chiamata di procedura e la gestione di array lineari.
- Ogni istruzione può essere preceduta da una o più etichette (stringhe letterali), utilizzate nelle istruzioni di salto.

Esempio (cfr: Compilatori: Principi, tecniche e strumenti)

- L'istruzione condizionale

```
if( x < 100 || x > 200 && x != y ) x = 0;
```

potrebbe essere tradotta nel seguente frammento di three address code:

```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:     x = 0
L1:
```

- Come si vede dall'esempio, il codice è sufficientemente vicino ad un ragionevole codice macchina, pur essendo indipendente da ogni particolare architettura.

Three address code (2)

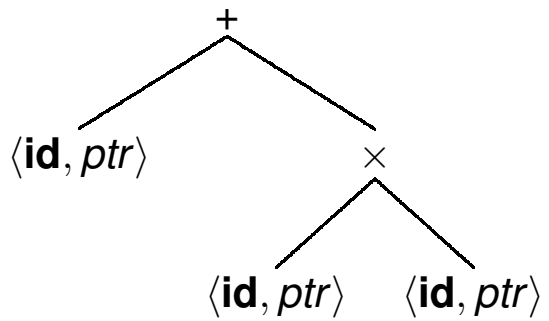
- I moderni compilatori generano codice intermedio lineare (come il three address code, del quale torneremo ad occuparci più avanti) in maniera diretta.
- In queste note (per ragioni didattiche) supporremo invece che il three address code sia il risultato finale di una serie di passaggi "più fini".
- Tali ulteriori passaggi intermedi trasformano il programma sorgente in rappresentazioni ad allbero equivalenti.
- Queste rappresentazioni sono lo stesso parse tree e, soprattutto, l'abstract syntax tree.
- Peraltro, la generazione esplicita del parse tree è (quasi sempre) evitabile.

Abstract syntax tree

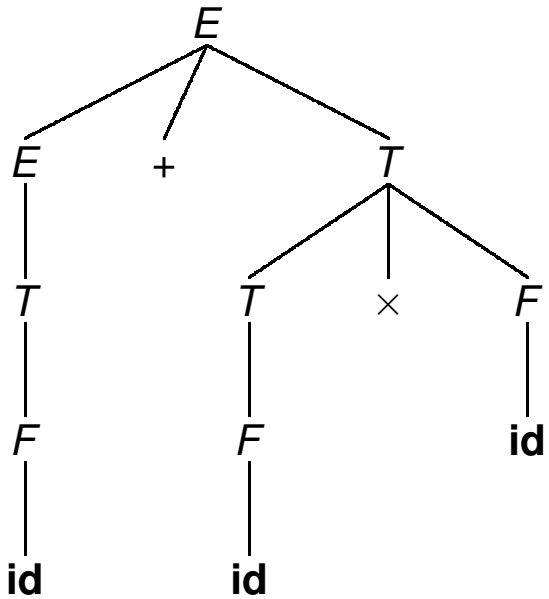
- Un *abstract syntax tree* (*AST*) per un linguaggio *L* è un albero in cui:
  - i nodi interni rappresentano costrutti di *L*;
  - i figli di un nodo che rappresenta un costrutto $\mathcal{C}$ rappresentano a loro volta le "componenti significative" di $\mathcal{C}$;
  - le foglie sono "costrutti elementari" (non ulteriormente decomponibili) caratterizzati da un *valore lessicale* (tipicamente un numero o un puntatore alla symbol table).
- Le diapositive seguenti illustrano la nozione di abstract syntax tree.

Esempio

- Un abstract syntax tree per la frase **id** $+$ **id** $\times$ **id** è

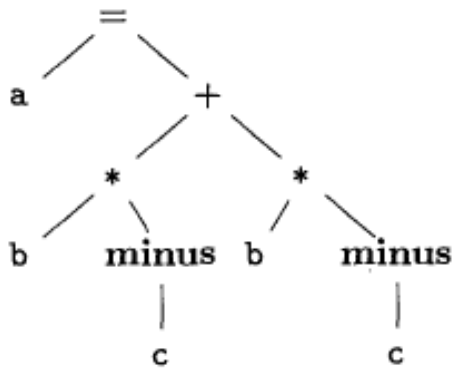


- Abstract syntax tree e parse tree sono oggetti diversi.



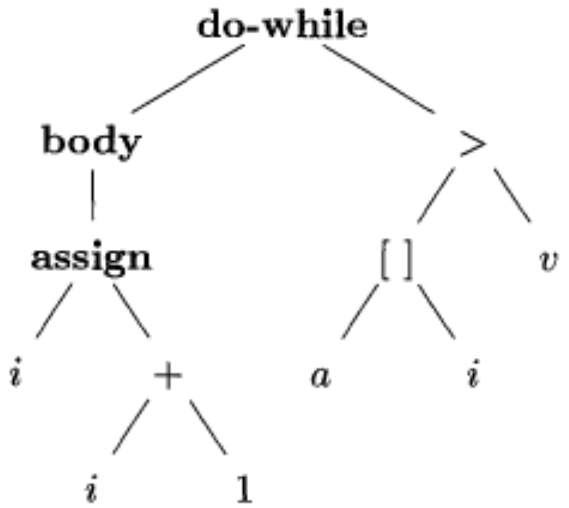
Esempio

- Un AST per l'assegnamento `a=b*(-c)+b*(-c)`:



- Un AST per il comando

do `i=i+1` **while** (`a[i]>v`)



Abstract syntax tree (2)

- L'utilità degli AST è riassumibile nelle seguenti affermazioni.
  - Partendo da un AST la generazione del three address code è un esercizio sufficientemente semplice (anche se l'intero processo è meno efficiente della generazione diretta di codice intermedio).
  - Nella realizzazione di semplici linguaggi intepretati (o comunque di applicazioni dove l'efficienza non sia il principale requisito) gli AST possono rappresentare il risultato ultimo della compilazione.
  - Risulta infatti molto semplice (in generale, e in rapporto alla complessità di realizzare un compilatore completo) implementare un software per interpretare gli AST.

Un semplice interprete per le espressioni aritmetiche

- La diapositiva seguente presenta lo pseudocodice per un semplice interprete di AST che rappresentano espressioni aritmetiche.
- Ogni nodo dell'albero tre campi:
  - un campo etichetta (`label`) che, se il nodo è interno, contiene un codice di operatore (come, ad esempio, `PLUS, TIMES, MINUS, UNARY_MINUS, ...`), se invece il nodo è una foglia contiene un puntatore alla symbol table;
  - un campo puntatore al primo operando (`left`);
  - un campo puntatore all'eventuale secondo operando.
- Lo pseudocodice usa una routine (`apply`) che restituisce il valore dell'applicazione di un operatore binario a due operandi passati come parametri.

Un semplice interprete per le espressioni aritmetiche

```
EVAL(NODE v)
1: if left(v) ≠ nil then
2:     x ← eval(left(v))
3:     if label(v) = UNARY_MINUS then
4:         return −x
5:     y ← eval(right(v))
6:     return apply(label(v), x, y)
7: else
8:     return symlookup(label(v))
```

Assunzioni

- Presentiamo ora alcuni semplici esempi di generazione di codice intermedio.
- Non intendiamo presentare una descrizione completa, sia per ragioni di tempo sia perché (a nostro avviso) questa attività è molto meno interessante dal punto di vista delle idee sottostanti.
- Gli esempi hanno dunque il solo scopo di illustrare l'approccio base per la generazione di codice intermedio, ignorando quegli aspetti che richiederebbero un ragionamento più approfondito.
- Faremo inoltre tre ipotesi molto forti, e precisamente: (1) di disporre dell'abstract syntax tree delle stringhe da tradurre; (2) che il risultato debba essere presentato sotto forma di file alfanumerico, (3) che esista un unico tipo di dato nei programmi sorgenti (ad esempio, numeri interi).

Istruzioni specifiche del three-address code

- Allo scopo di presentare gli esempi, considereremo solo le seguenti istruzioni, il cui significato dovrebbe risultare chiaro:
  - $A \leftarrow B$ `op` C, dove `op` $\in \{+, -, \times, /\}$ e A, B e C sono identificatori definiti dall'utente nel programma sorgente oppure temporanee generate dal parser;
  - $A \leftarrow B$, dove A e B sono definiti come al punto precedente;
  - $A \leftarrow$ `op` B, dove `op` è un operatore unario;
  - `goto` L, dove L è un'etichetta generata dal parser;
  - `if` A `goto` L, dove A è un identificatore definito dall'utente oppure una temporanea generata dal parser ed L è un'etichetta generata dal parser;
  - `ifFalse` A `goto` L, dove A ed L sono definiti come al punto precedente.

Ulteriori assunzioni

- Ipotizzeremo che il parser possa invocare una funzione per generare identificatori univoci, come pure una funzione per generare etichette univoche.
- Assumeremo inoltre la disponibilità di una funzione, che chiameremo *emit*(), che stampa la stringa passata come parametro (che rappresenta una porzione del programma in three-address code) su un opportuno supporto di output.
- Assumeremo infine che il generico nodo dell'abstract syntax tree abbia la seguente struttura:
  - un campo `label` che caratterizza il tipo di nodo (assegnamento, operatore, comando if, ...);
  - se significativo (ad esempio nel caso di identificatore o operatore), un puntatore alla symbol table per il corrispondente valore lessicale, accessibie mediante la funzione *lexval*;
  - uno o più puntatori ai nodi che rappresentano i componenti del costrutto, accessibili mediante i campi `c1`, `c2`, ...

La struttura generale del traduttore

- Con le ipotesi fatte, il traduttore (da syntax tree a three-address code) può essere organizzato come procedura ricorsiva il cui corpo è costituito essenzialmente da una "grossa" istruzione `case` (o, se si preferisce, `switch`), che analizza il nodo `p` passato come parametro e, a seconda del tipo, esegue operazioni diverse.
- Data la struttura del parse tree, la generazione del codice per un dato nodo implicherà poi una o più chiamate ricorsive per la generazione del codice associato ai nodi figli.
- La procedura, che chiameremo *gencode*, riceve un ulteriore parametro (*RES*) che è una stringa (eventualmente vuota) alla quale (vista come nome di variabile) deve essere assegnato il risultato calcolato dal codice generato per il nodo *p*.

Procedura gencode

Procedure 1 *gencode*(*string* `RES`, *AST** `p`)

```
tag ← (p → label)
case tag of
    id :    ...
    number :    ...
    assignment :    ...
    comparison :    ...
    binaryop :    ...
    unaryminus :    ...
    seq :    ...
    if :    ...
    ifElse :    ...
    while :    ...
...
    default :    error()
end
```

Il caso degli identificatori

- Si tratta del caso più semplice da trattare.
- Infatti, se un nodo è etichettato come identificatore, tutto ciò che bisogna fare è semplicemente emettere una stringa che ne rappresenta il valore lessicale.
- Al riguardo, utilizziamo una funzione *toString* (esiste anche in Java), che, nel caso il valore lessicale dell'identificatore sia già internamente rappresentato come stringa, equivale ad un no-op.
- Per altri tipi di nodo, *toString* svolge effettivamente una funzione: ad esempio, se il nodo è un operatore binario, la chiamata $toString(lexval(p))$ ne fornisce la consueta rappresentazione matematica (+,-, ecc).

Il caso degli identificatori (continua)

- Il codice relativo a questo caso è dunque:

  id : `string name` $\leftarrow$ *toString*(*lexval*(*p*))
  if not *empty*(`RES`) **then**
  emit(`RES`+"$\leftarrow$"+`name`)
  else
  emit(`name`)

  dove l'operatore + applicato a stringhe denota concatenazione.
- Si noti anche il controllo (che sarà ricorrente anche nei seguenti esempi) sulla stringa `RES`.
- Se `RES` non è la stringa vuota, il codice da generare deve infatti prevedere un assegnamento al nome da essa rappresentato.

Costanti numeriche

- Il caso delle costanti numeriche è identico a quello degli identificatori.
- C'è solo un maggior lavoro (nascosto) da parte della procedura `toString`, che deve ri-trasformare in sequenza ASCII un numero rappresentato internamente in virgola fissa o virgola mobile.

id : `string const` $\leftarrow$ *toString*(*lexval*(*p*))
if not *empty*(`RES`) **then**
    emit(`RES`+"$\leftarrow$"+`const`)
else
    emit(`const`)

Assegnamento

- Un nodo etichettato come *assignment* ha due figli, il primo dei quali deve necessariamente essere un *id*.
- Il codice da generare prevede la chiamata ricorsiva al secondo figlio, in modo che lasci il valore nella variabile il cui nome è il valore lessicale del primo figlio.
- In altre parole:

assignment : `string name` $\leftarrow$ *lexval*(`p` $\rightarrow$ `c1`)
if not *empty*(`RES`) **then**
    gencode(`name`, `p` $\rightarrow$ `c2`)
    emit(`RES`+"$\leftarrow$"+*name*)
else
    gencode(`name`, `p` $\rightarrow$ `c2`)

Gli operatori (aritmetici) binari

- Se l'etichetta del nodo è un operatore binario bisogna:
  - generare ricorsivamente il codice per il figlio di sinistra, in modo che lasci il risultato in una variabile il cui nome univoco è generato dal parser stesso (supporremo che tali nomi abbiano la forma `TEMP`*n*, con *n* progressivo);
  - generare (analogamente) il codice per figlio di destra, in modo che lasci il risultato in una seconda variabile;
  - generare la stringa per un'istruzione a tre o due indirizzi (a seconda del valore del parametro `RES`) che esegue l'operazione indicata dal particolare operatore binario sui dati memorizzati nelle temporanee.

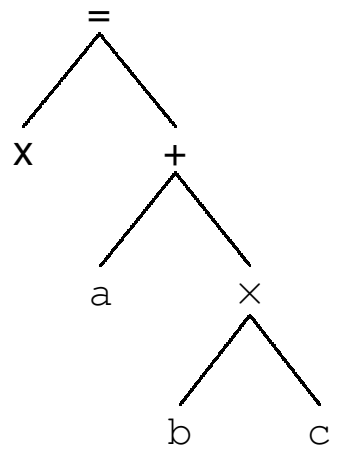
Gli operatori (aritmetici) binari (continua)

- Il codice corrispondente è:

  binaryop :
  `string t1` $\leftarrow$ *new temporary*()
  `string t2` $\leftarrow$ *new temporary*()
  gencode(`t1`, `p` $\rightarrow$ `c1`)
  gencode(`t2`, `p` $\rightarrow$ `c2`)
  `string op` $\leftarrow$ *toString*(*lexval*(*p*))
  if not *empty*(`RES`) **then**
  emit(`RES`+"$\leftarrow$"+`t1` + `op` + `t2`)
  else
  emit(`t1` + `op` + `t2`)

Esempio

- Per l'abstract syntax tree del comando C/C++ `x = a + b * c` (rappresentato in modo da evidenziare direttamente il valore lessicale degli operatori e degli identificatori)



viene generato il seguente codice a tre indirizzi

```
temp1 ← a
temp3 ← b
temp4 ← c
temp2 ← temp3 * temp4
x ← temp1 + temp2
```

L'operatore "meno" unario

- È semplicissimo da realizzare.
- Si tratta dapprima di generare il codice per l'espressione che costituisce l'unico figlio, lasciando il risultato il una temporanea.
- Al risultato si applica poi l'operatore meno unario `uminus`.
- Il codice è:

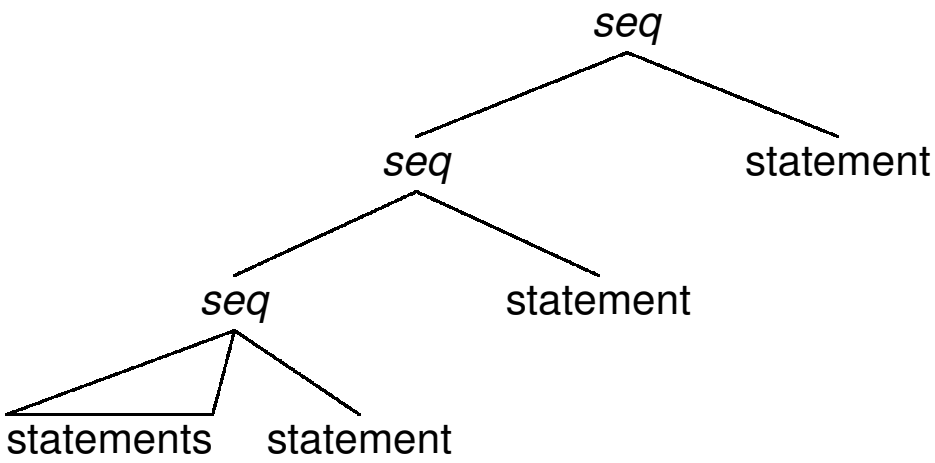
unaryminus :
`string t` $\leftarrow$ *new temporary*()
gencode(`t,p` $\rightarrow$ `c1`)
if not *empty*(`RES`) **then**
    emit(`RES`+"$\leftarrow$ `uminus`"+`t`)
else
    emit("`uminus` + `t`")

Sequenza

- Una sequenza di comandi, definita dalle produzioni

$$L \rightarrow L; S \mid S,$$

produce alberi sintattici con la struttura indicata di seguito (in cui ogni singolo statement può, a sua volta, essere un syntax tree)

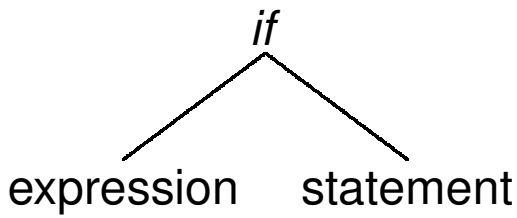


- Il codice corrispondente consiste semplicemente di due chiamate ricorsive:

seq :
gencode(“”, p → c1)
gencode(“”, p → c2)

Comando “If then”

- È rappresentato da alberi sintattici del tipo



- I passi per effettuare la traduzione sono i seguenti:
  - si genera ricorsivamente il codice per calcolare l’espressione, lasciando il risultato in una temporanea;
  - si genera un’etichetta e si emette un’istruzione di salto a tale etichetta, condizionato al valore falso della temporanea;
  - si genera quindi ricorsivamente il codice per il comando (che verrà dunque eseguito se il valore della temporanea è vero);
  - infine si emette l’etichetta generata (che andrà ad etichettare la prossima istruzione a tre indirizzi, non emessa dal trattamento del condizionale).

Comando “If then” (2)

- Il codice corrispondente è:

  if :
  `string t` $\leftarrow$ *new temporary*()
  gencode(`t,p` $\rightarrow$ `c1`)
  `string l` $\leftarrow$ *new label*()
  emit(“`ifFalse `”+`t`+“ `goto `”+`l`)
  gencode(“”,`p` $\rightarrow$ `c2`)
  emit(`l`+“: ”)

Comando “If then else”

- È solo leggermente più complicato del caso precedente, per cui presentiamo direttamente il codice

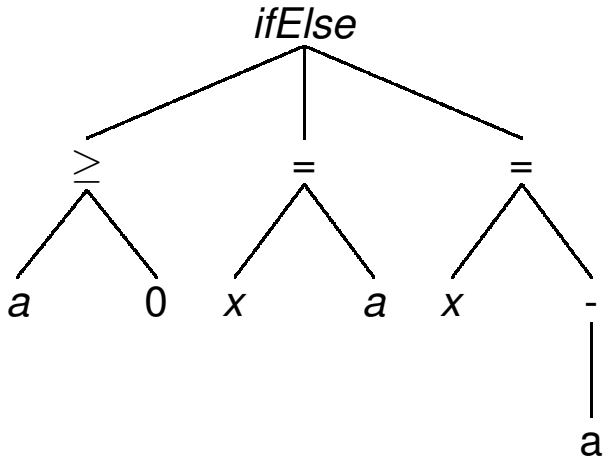
  ifElse :
  `string t` $\leftarrow$ *new temporary*()
  gencode(`t,p` $\rightarrow$ `c1`)
  `string l1` $\leftarrow$ *new label*()
  emit(“`ifFalse `”+`t`+“ `goto `”+`l1`)
  gencode(“”,`p` $\rightarrow$ `c2`)
  `string l2` $\leftarrow$ *new label*()
  emit(“`goto `”+`l2`)
  emit(`l1` + “ :”)
  gencode(“”,`p` $\rightarrow$ `c3`)
  emit(`l2`+“: ”)

Esempio

- Alla frase C/C++

$$\texttt{if } a >= 0 \texttt{ then } x = a \text{ else } x = -a$$

corrisponde il seguente abstract syntax tree



(si ricordi che abbiamo scelto di evidenziare direttamente il valore lessicale degli operatori e degli identificatori anziché inserire simboli astratti e riferimenti alla symbol table).

Esempio (continua)

- Il codice a tre indirizzi corrispondente è:

```
temp2 ← a
temp3 ← 0
temp1 ← temp2 ≥ temp3
ifFalse temp1 goto label1
x ← a
goto label2
label1 : temp4 ← a
x ← −temp4
label2 :
```

Comando “while”

- Come ultimo caso, consideriamo la traduzione di abstract syntax tree corrispondenti al costrutto while.
- Il costrutto ha due componenti, la condizione e lo statement da ripetere finché la condizione è vera.
- La strategia di traduzione consiste quindi nel generare il codice per la condizione, emettere un’istruzione di salto condizionato (`ifFalse`), generare il codice per il comando ed emettere un’istruzione di salto incondizionato al codice generato per la condizione.
- Lo pseudocodice dettagliato è riportato nella successiva trasparenza.

Comando “while” (2)

while :
`string t` $\leftarrow$ *new temporary*()
`string l1` $\leftarrow$ *new label*()
emit(`l1`+“: ”)
gencode(`t`, `p` $\rightarrow$ `c1`)
`string l2` $\leftarrow$ *new label*()
emit(“`ifFalse `”+`t`+“ `goto `”+`l2`)
gencode(“”, `p` $\rightarrow$ `c2`)
emit(“`goto `”+`l1`)
emit(`l2`+“: ”)

Traduzione guidata dalla sintassi

- La generazione dell'abstract syntax tree, il nostro obiettivo attuale, può essere eseguita mediante l'applicazione di una tecnica nota come *Traduzione guidata dalla sintassi* (in inglese *Syntax-Directed Translation*).
- L'output desiderato (un abstract syntax tree) viene cioè generato mediante un processo che è guidato dalla stessa grammatica e, in particolare, dalle sue produzioni.
- La tecnica è simile a quella utilizzata da Yacc per associare azioni alle produzioni.
- Nel caso di Yacc le azioni potevano essere di natura qualsiasi, mentre qui ci interesseranno solo azioni di costruzione dell'abstract syntax tree.

Traduzione guidata dalla sintassi (2)

- Tipicamente, quando viene effettuata una riduzione (nel parsing bottom-up, come per Yacc), viene anche eseguita una qualche azione (ad esempio, la creazione di nodi e il loro "aggancio" all'albero in costruzione).
- Nel caso di parsing top-down l'azione viene eseguita in fase di applicazione di una produzione.
- La modalità specifica che vedremo per la realizzazione del processo di costruzione dell'AST è quella degli *schemi di traduzione* (*syntax-directed translation scheme*).

Syntax-directed translation scheme

- Uno schema di traduzione prevede la specifica di frammenti di codice, che vengono associati alle produzioni della grammatica (esattamente come avviene in Yacc)
- L'insieme dei frammenti non costituisce però un programma
- Al riguardo, e sempre ponendo a mente il caso di Yacc, ci sono due fondamentali domande cui dobbiamo rispondere
  1. Quali sono e dove sono memorizzati i dati manipolati dai frammenti di codice?
  2. Chi e come "forza" un ordine di esecuzione dei frammenti in modo che costituiscano un programma organico e corretto?

Attributi

- Nel caso degli schemi di traduzione, i dati manipolati sono *attributi* associati ai simboli della grammatica.
- Se X è un simbolo della grammatica e a è il nome di un suo attributo, useremo la scrittura $X.a$ per indicare il valore dell'attributo a di X.
- Come già osservato a proposito di Yacc, la scrittura $X.a$ si applica non tanto al "generico" simbolo X bensì ad una sua occorrenza nella derivazione (o, equivalentemente, nel parse tree).
- Il valore di un attributo può essere di qualunque natura: stringa, numero, puntatore, tipo di dato, ecc.

Attributi (2)

- Nel nostro caso, i valori degli attributi saranno puntatori al "costruendo" syntax tree.
- Per questa ragione, associato ad ogni non terminale della grammatica si troverà sempre un attributo di nome *node*.
- In alcuni casi ci potranno essere anche ulteriori attributi
- Siamo ora in grado di presentare il nostro primo esempio di schema di traduzione, relativo alla più semplice grammatica per le espressioni aritmetiche.

Esempio

$$
\begin{array}{lcll}
E & \to & E_1 + T & \{E.node \leftarrow \texttt{new node}('+', E_1.node, \\
 & & & \quad T.node)\} \\
E & \to & T & \{E.node \leftarrow T.node\} \\
T & \to & T_1 \times F & \{T.node \leftarrow \texttt{new node}('\times', T_1.node, \\
 & & & \quad F.node)\} \\
T & \to & F & \{T.node \leftarrow F.node\} \\
F & \to & (E) & \{F.node \leftarrow E.node\} \\
F & \to & \mathbf{id} & \{F.node \leftarrow \texttt{new leaf}(\mathbf{id}, \mathbf{id}.lexval)\}
\end{array}
$$

- Si noti che:
  - ai non terminali è associato il solo attributo *node*;
  - ai terminali è associato l'attributo *lexval* (tipicamente, un puntatore alla entry della symbol table).
  - l'aggiunta dell'indice ai non terminali E e T, nella parte destra delle produzioni $E \to E + T$ e $T \to T \times F$, serve unicamente a disambiguare i riferimenti nelle corrispondenti azioni.

Calcolabilità degli schemi

- Come possiamo acquisire la capacità di scrivere schemi di traduzione corretti?
- In parole ancora più semplici: come possiamo imparare a mettere le “azioni giuste” a fianco delle produzioni per poter creare effettivamente il syntax tree della frase in input?
- Per raggiungere questo obiettivo, bisogna avere chiaro come il parser *mette assieme* le singole azioni specificate in un processo di calcolo completo.
- Al riguardo è utile immaginare che il parser costruisca effettivamente il parse tree.

Calcolabilità degli schemi (2)

- L'utilità del parse tree è che esso rende più facile visualizzare istanze diverse di uno stesso simbolo della grammatica.
- Ad esempio, se il non terminale E viene usato due volte in una derivazione, questo appare in modo evidente nel parse tree.
- Ci saranno infatti due nodi etichettati con il simbolo E.
- Se riferita al parse tree, un'azione come $\{T.node \leftarrow F.node\}$ "dice" che il parser dovra copiare il valore di un attributo (*node*) da un nodo ad un altro dell'albero.

Calcolabilità degli schemi (3)

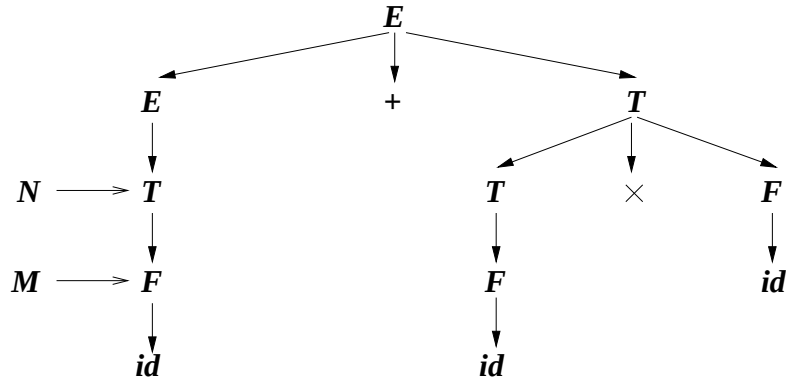
- Utilizzando il parse tree, inoltre, l'ordine di esecuzione delle azioni di uno schema è determinato da da un processo (ideale) di attraversamento del parse tree.
- L'ordine di visita (e dunque di esecuzione delle istruzioni previste nello schema) deve essere tale che il valore calcolato ad un dato nodo dipenda solo da valori che sono già stati calcolati (dunque in nodi già visitati).

Calcolabilità degli schemi (4)

- Riconsideriamo lo schema relativo alla semplice grammatica le espressioni:

$$
\begin{array}{lcll}
E & \rightarrow & E_1 + T & \{E.node \leftarrow \texttt{new node}('+', E_1.node, T.node)\} \\
E & \rightarrow & T & \{E.node \leftarrow T.node\} \\
T & \rightarrow & T_1 \times F & \{T.node \leftarrow \texttt{new node}('\times', T_1.node, F.node)\} \\
T & \rightarrow & F & \{T.node \leftarrow F.node\} \\
F & \rightarrow & (E) & \{F.node \leftarrow E.node\} \\
F & \rightarrow & \mathbf{id} & \{F.node \leftarrow \texttt{new leaf}(\mathbf{id}, \mathbf{id}.lexval)\}
\end{array}
$$

e il parse tree relativo alla frase **id** $+$ **id** $\times$ **id**:



Calcolabilità degli schemi (5)

- Nel parse tree soffermiamoci su due nodi N ed M (padre e figlio) etichettati rispettivamente con i non terminali T ed F.
- In corrispondenza della produzione $T \rightarrow F$ è presente la regola $\{T.node \leftarrow F.node\}$.
- Questo richiede che M sia visitato prima di N.
- In generale le azioni eseguita dal parser saranno quantomeno coerenti se inserite in una visita dell'albero che soddisfa la seguente *proprietà fondamentale*:

  > se un attributo $X.a$ presente in un nodo N (con etichetta X) dipende da un attributo $Y.b$ presente in un nodo M, allora M viene (almeno parzialmente) visitato prima di N.

Tipi di attributi

- Definiamo le due diverse tipologie di attributi utilizzate nelle applicazioni reali: gli attributi *sintetizzati* e gli attributi *ereditati*.
- Consideriamo una generica produzione

$$A \to X_1 \dots X_{i-1} X_i \dots X_k$$

- un attributo di A si dice *sintetizzato* se il suo valore dipende solo dal valore degli attributi dei simboli X_i nella parte destra della produzione e (eventualmente) di altri attributi di A stesso.
- In termini di parse tree, è sintetizzato un attributo nel nodo A che dipenda solo dagli attributi in A o nei nodi figli $X_1 \dots X_{i-1} X_i \dots X_k$.

Tipi di attributi (2)

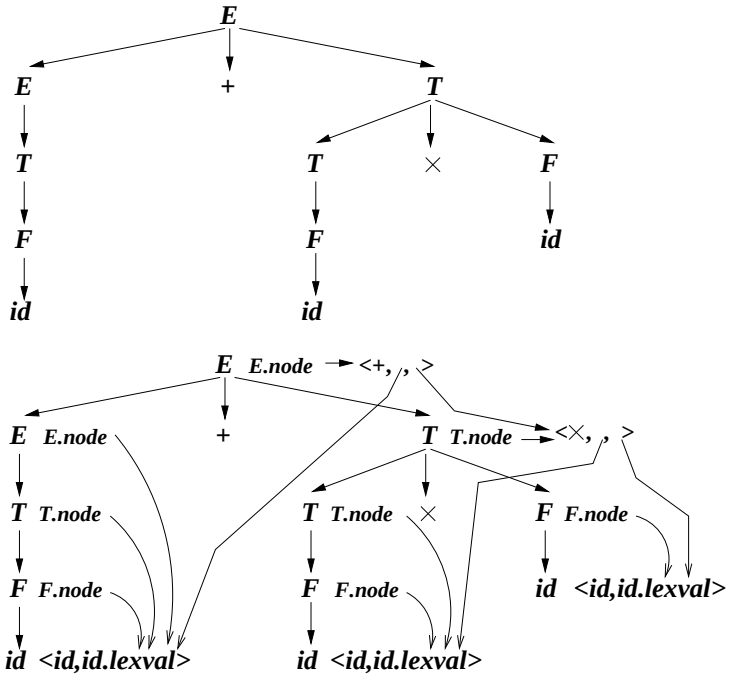
- I simboli terminali della grammatica avranno solo attributi sintetizzati.
- Uno schema in cui tutti gli attributi siano sintetizzati è detto *S-attributed*.
- Lo schema:

$$
\begin{array}{lcll}
E & \rightarrow & E_1 + T & \{E.node \leftarrow \texttt{new node}('+', E_1.node, \\
 & & & \quad T.node)\} \\
E & \rightarrow & T & \{E.node \leftarrow T.node\} \\
T & \rightarrow & T_1 \times F & \{T.node \leftarrow \texttt{new node}('\times', T_1.node, \\
 & & & \quad F.node)\} \\
T & \rightarrow & F & \{T.node \leftarrow F.node\} \\
F & \rightarrow & (E) & \{F.node \leftarrow E.node\} \\
F & \rightarrow & \mathbf{id} & \{F.node \leftarrow \texttt{new leaf}(\mathbf{id}, \mathbf{id}.lexval)\}
\end{array}
$$

è chiaramente S-attributed.

Esempio

- Esaminando la seguente figura, si comprende come l'attraversamento in ordine posticipato del parse tree per **id** $+$ **id** $\times$ **id** porti alla costruzione dell'AST.



- Si tenga presente che la visita di un nodo consiste nel calcolo dei suoi attributi (secondo lo schema).

Tipi di attributi (3)

- Consideriamo ancora la generica produzione

$$A \to X_1 \dots X_{i-1} X_i \dots X_k$$

- Un attributo per X_i si dice *ereditato* se il suo valore dipende (eventualmente) dal valore di attributi ereditati di A e di attributi (ereditati o sintetizzati) di $X_1, \dots, X_{i-1}$.
- In termini di parse tree, un attributo a un nodo è ereditato se il suo valore dipende dal valore di attributi ereditati al nodo genitore o attributi (di qualunque tipo) ai nodi che sono “fratelli maggiori”.
- Questa definizione non è la più generale ma, di fatto, è quella operativa.

Tipi di attributi (4)

- Schemi di traduzione in cui vi sia (oltre ad eventuali attributi sintetizzati) almeno un attributo ereditato secondo questa definizione vengono detti *L-attributed*.
- Lo schema di traduzione:

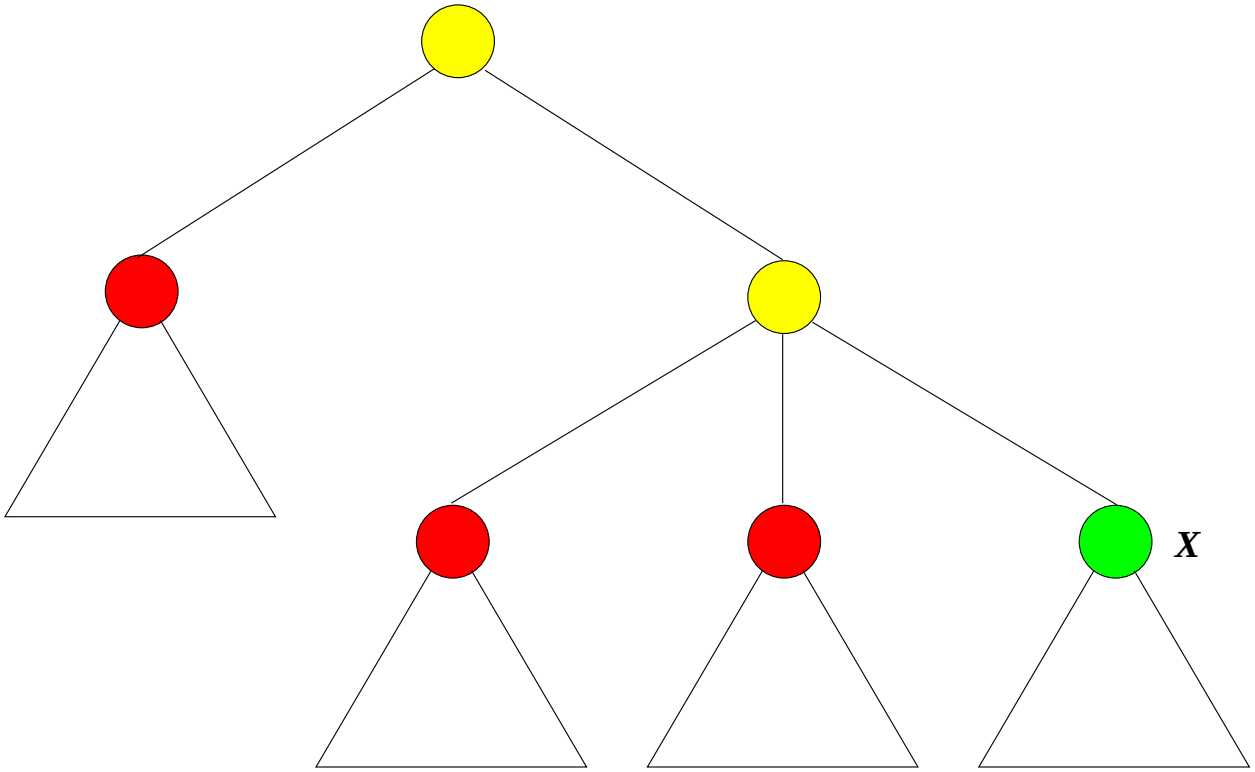
$$
\begin{array}{lcll}
E & \rightarrow & TE' & \{E'.inh \leftarrow T.node;\ E.node \leftarrow E'.node\} \\
E' & \rightarrow & +TE'_1 & \{E'_1.inh \leftarrow \texttt{new node}('+', E'.inh, T.node); \\
 & & & \quad E'.node \leftarrow E'_1.node\} \\
E' & \rightarrow & \epsilon & \{E'.node \leftarrow E'.inh\} \\
T & \rightarrow & FT' & \{T'.inh \leftarrow F.node;\ T.node \leftarrow T'.node\} \\
T' & \rightarrow & \times FT'_1 & \{T'_1.inh \leftarrow \texttt{new node}('\times', T'.inh, F.node); \\
 & & & \quad T'.node \leftarrow T'_1.node\} \\
T' & \rightarrow & \epsilon & \{T'.node \leftarrow T'.inh\} \\
F & \rightarrow & (E) & \{F.node \leftarrow E.node\} \\
F & \rightarrow & \mathbf{id} & \{F.node \leftarrow \texttt{new leaf}(\mathbf{id}, \mathbf{id}.lexval)\}
\end{array}
$$

relativo alla grammatica per le espressioni adatta al top-down parsing è L-attributed.

Tipi di attributi (5)

- Gli attributi ereditati possono essere calcolati nell'ambito di una visita in ordine anticipato del parse tree.
- Infatti, nel momento in cui viene calcolato il valore di un attributo ereditato ad un nodo X:
  - i nodi che sono "fratelli maggiori" di X sono stati già visitati e i loro attributi (siano essi sintetizzati o ereditati) già calcolati;
  - gli attributi sintetizzati al nodo genitore non sono stati calcolati ma (con ragionamento ricorsivo) quelli ereditati sì.
- La slide successiva illustra graficamente la validità delle precedenti osservazioni.

Tipi di attributi (6)



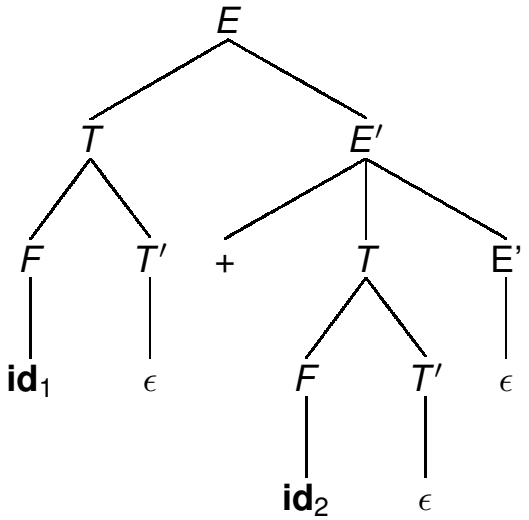
- Ai nodi colorati di rosso tutti gli attributi sono stati calcolati.
- Ai nodi colorati di giallo sono stati calcolati solo gli attributi ereditati.
- Al nodo colorato di verde (il “nostro” nodo X) gli attributi non sono ancora stati calcolati.

Tipi di attributi (7)

- In uno schema L-attributed che abbia anche attributi sintetizzati, il calcolo deve essere effettuato sia in fase di discesa nel parse tree, sia in fase di risalita.
- Più precisamente, ad ogni dato nodo X gli attributi ereditati vengono calcolati prima di visitare i figli di X mentre gli attributi sintetizzati vengono calcolati dopo.
- Ad esempio, se il processo è realizzato mediante una procedura ricorsiva, il calcolo degli attributi ereditati avviene prima delle chiamate ricorsive sui nodi figli, mentre quello degli attributi sintetizzati avviene al ritorno di tali chiamate.

Esempio

- Per illustrare il processo generale ci aiutiamo con il semplice esempio di parsing della frase $\mathbf{id}_1 + \mathbf{id}_2$:



- Come già per gli schemi S-attributed, le dipendenze funzionali fra gli attributi (e dunque il flusso di dati necessario affinché il calcolo proceda in modo corretto) possono essere evidenziati utilizzando un parse tree "annotato".

Esempio (continua)

- Per descrivere il flusso di informazione utilizziamo, nella figura usiamo le seguenti convenzioni:
  - una freccia ——► indica un arco del parse tree (queste frecce sono sempre orientate verso il basso);
  - una freccia ——➢ (orientata verso l'alto) indica che l'attributo sintetizzato al nodo di partenza è usato per calcolare l'attributo sintetizzato al nodo di arrivo;
  - una freccia ——○ (orientata verso il basso) indica che l'attributo ereditato al nodo di partenza è usato per calcolare l'attributo ereditato al nodo di arrivo;
  - infine, una freccia ——● (orientata verso destra) indica che l'attributo sintetizzato al nodo di partenza è usato per calcolare l'attributo ereditato al nodo di arrivo.

Esempio (continua)

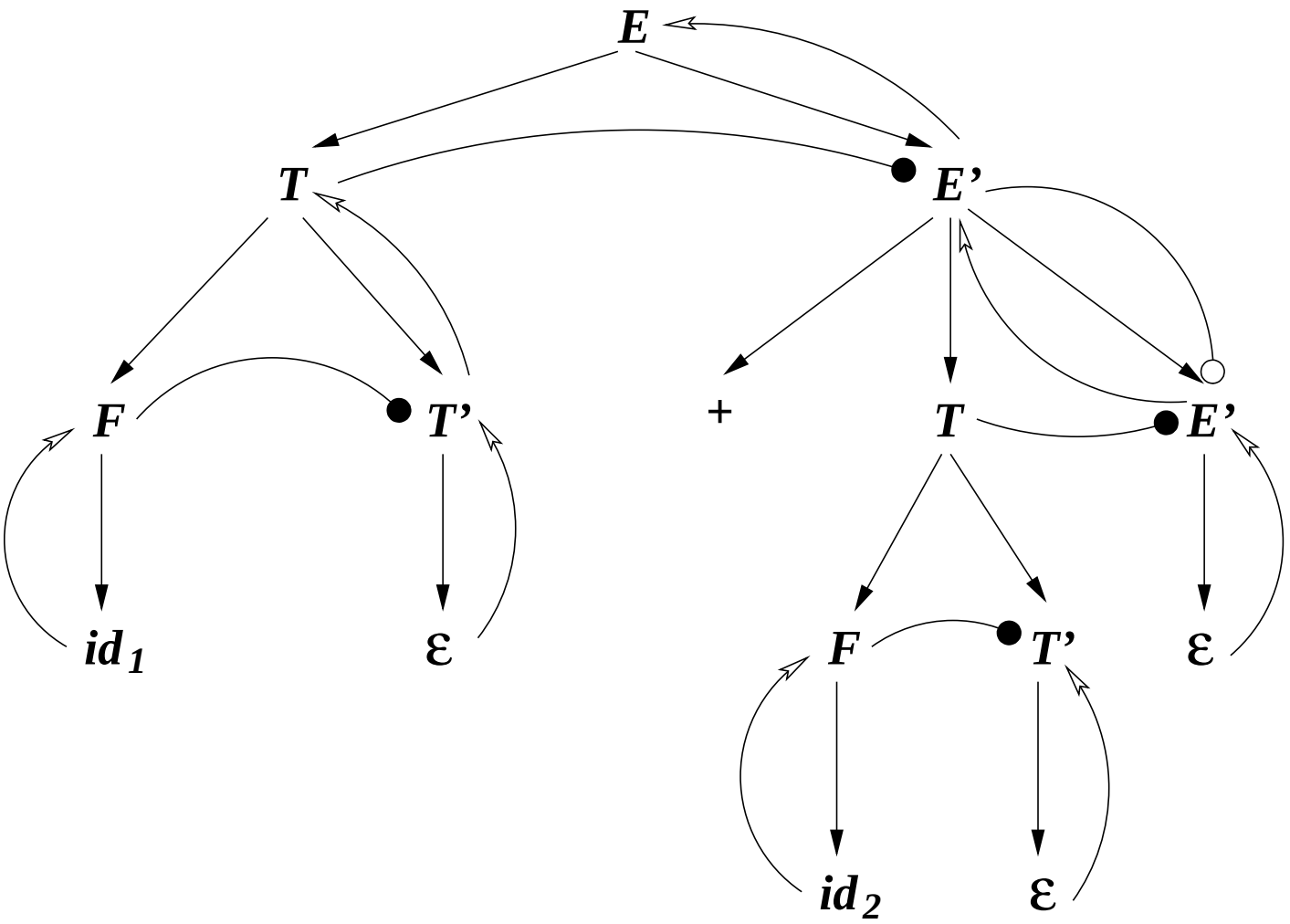
- Riportiamo nuovamente lo schema di traduzione:

$$
\begin{array}{lcll}
E & \to & TE' & \{E'.inh \leftarrow T.node;\ E.node \leftarrow E'.node\} \\
E' & \to & +TE'_1 & \{E'_1.inh \leftarrow \texttt{new node}('+', E'.inh, T.node);\ E'.node \leftarrow E'_1.node\} \\
E' & \to & \epsilon & \{E'.node \leftarrow E'.inh\} \\
T & \to & FT' & \{T'.inh \leftarrow F.node;\ T.node \leftarrow T'.node\} \\
T' & \to & \times FT'_1 & \{T'_1.inh \leftarrow \texttt{new node}('\times', T'.inh, F.node);\ T'.node \leftarrow T'_1.node\} \\
T' & \to & \epsilon & \{T'.node \leftarrow T'.inh\} \\
F & \to & (E) & \{F.node \leftarrow E.node\} \\
F & \to & \mathbf{id} & \{F.node \leftarrow \texttt{new leaf}(\mathbf{id}, \mathbf{id}.lexval)\}
\end{array}
$$

e, nella slide successiva, il parse tree annotato

Esempio (continua)

- Il parse tree annotato:



Schemi e tipo di parsing

- In un parser reale il parse tree non viene esplicitamente generato.
- Il soddisfacimento delle dipendenze fra attributi (e quindi la possibilità di avere un ordine di esecuzione “corretto” delle azioni previste nello schema) può comunque essere ottenuto almeno nei seguenti due casi:
  - la grammatica considerata ammette parsing di tipo LR e lo schema utilizzato è S-attributed, oppure
  - la grammatica ammette parsing di tipo LL e lo schema è L-attributed.

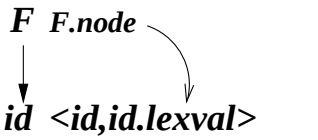
Parsing LR

- La validità della prima combinazione, di grammatica analizzabile con parser LR unitamente a schema S-attributed, è più semplice da dimostrare.
- Consideriamo infatti l'esecuzione di una riduzione in un parser *LR*, ad esempio $A \rightarrow XYZ$.
- Dimostriamo per induzione che (le occorrenze de) gli attributi dei simboli non terminali sono calcolati correttamente.
- È evidente che se tutti i simboli nella parte destra della produzione sono terminali (base), allora il calcolo degli eventuali attributi di A può essere eseguito correttamente.
- Se invece a destra vi è un simbolo non terminale, e supponiamo che Y lo sia, allora il parser ha già eseguito una riduzione $Y \rightarrow \alpha$ e dunque, per ipotesi induttiva, gli attributi di Y sono stati calcolati correttamente.

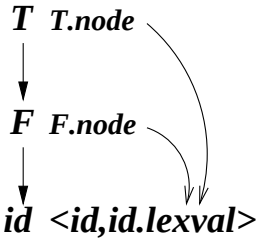
Esempio

- Riconsideriamo in dettaglio l'esempio relativo al parsing della stringa **id** $+$ **id** $\times$ **id**, secondo lo schema (S-attributed) introdotto.
- Le seguenti trasparenze illustrano come procede la computazione.
- Nelle illustrazioni il parse tree è mostrato solo allo scopo di evidenziare quali attributi sono utilizzati nei vari stadi della computazione.
- I nodi del parse tree, infatti, fornisce un "luogo" dove memorizzare tali attributi.
- In effetti, uno dei problemi che dovremo risolvere in assenza del parse tree è proprio capire "dove" memorizzare gli attributi.

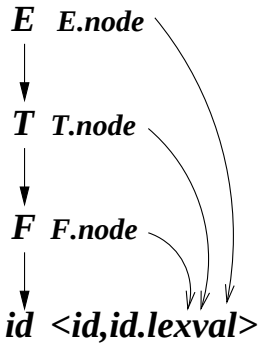
Esempio (continua)



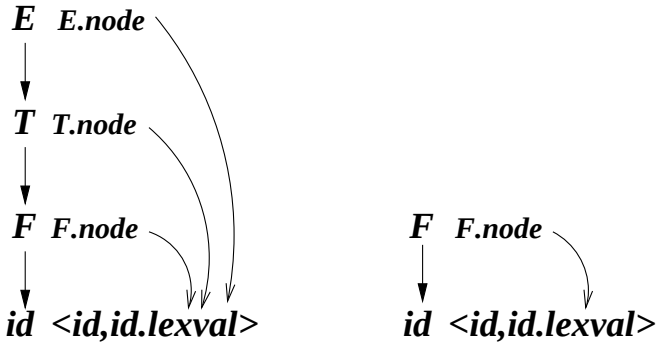
Dopo la riduzione $F \rightarrow id$



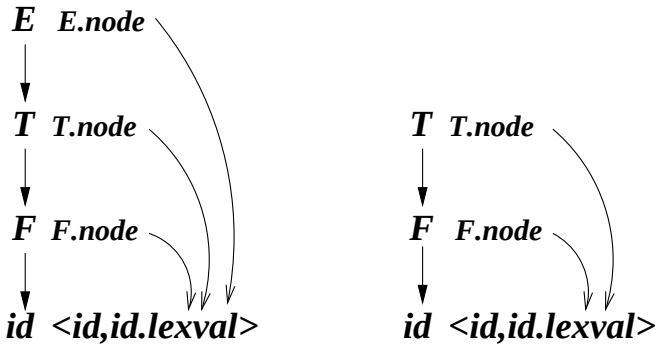
Dopo $T \rightarrow F$



Dopo $E \rightarrow T$

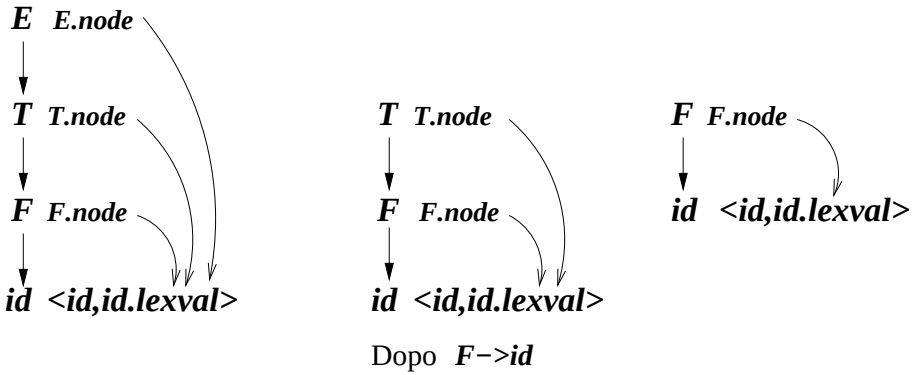


Dopo $F \rightarrow id$

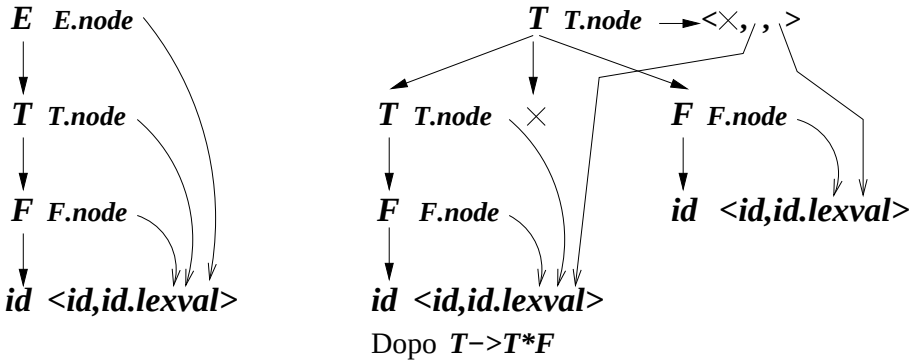


Dopo $T \rightarrow F$

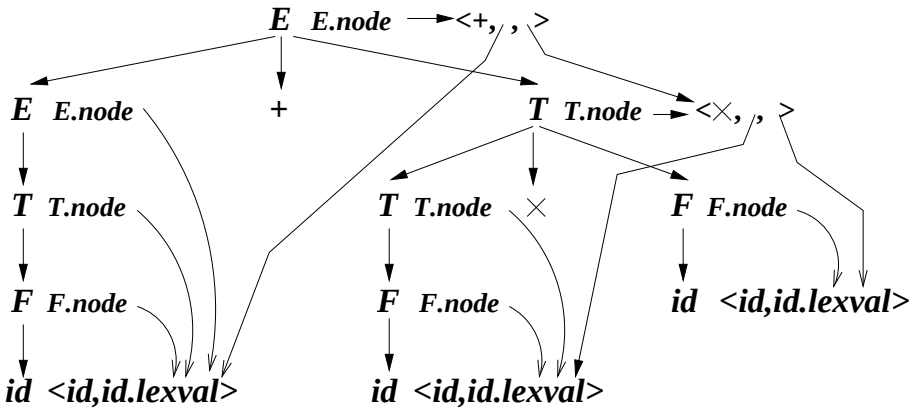
Esempio (continua)



Dopo $F \rightarrow id$



Dopo $T \rightarrow T*F$



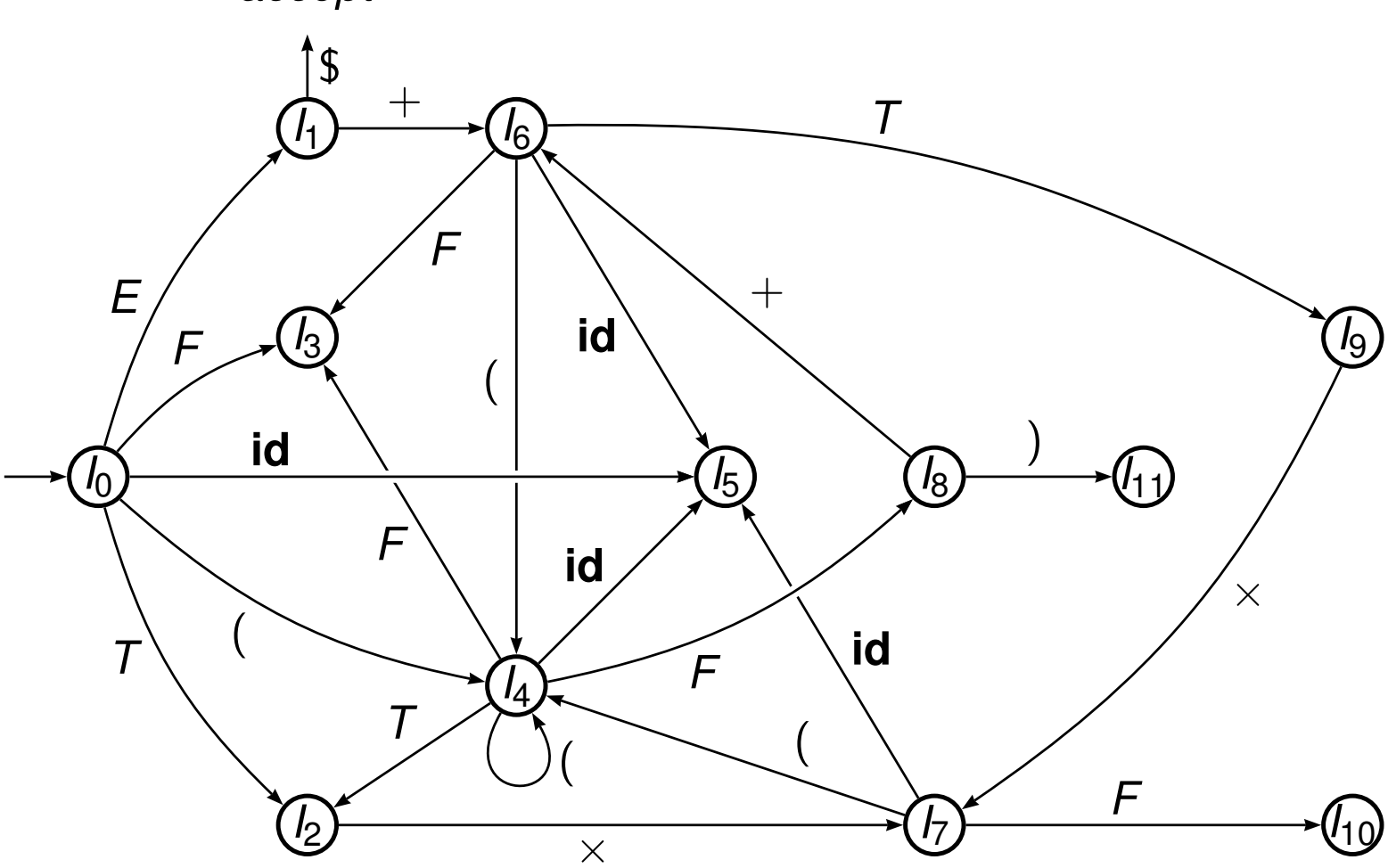
Dopo $E \rightarrow E+T$

Aspetti implementativi

- Nel caso di parsing LR e schema S-attributed è relativamente semplice mostrare come la costruzione esplicita del parse tree non sia richiesta.
- Osserviamo infatti che, non appena un nodo N viene collegato al padre P nel parse tree (a seguito di una riduzione), gli attributi definiti in N, e quindi anche N stesso, non servono più.
- È dunque sufficiente modificare lo stack utilizzato dal parser LR, in modo che esso memorizzi non solo stati ma anche gli attributi associati al simbolo che corrisponde a quello stato.
- A quest'ultimo riguardo, ricordiamo come ogni stato s dell'automa sia associato ad un simbolo della grammatica (precisamente quel simbolo che etichetta le transizioni entranti in s).

Aspetti implementativi (2)

- Si può verificare la validità dell'ultima affermazione nel caso dell'automa $LR(0)$ della grammatica LR(0) per le espressioni aritmetiche:



Schema di traduzione modificato

- Presentiamo di seguito lo schema di traduzione per la grammatica delle espressioni, modificato in modo da utilizzare lo stack per memorizzare gli attributi.
- Lo stack memorizza ora coppie $\langle s, p \rangle$, dove s è uno stato (un numero compreso fra 0 e 11), mentre p è il valore dell'attributo *node* (in pratica, un puntatore ad un nodo dell'abstract syntax tree in costruzione) del simbolo associato ad s.
- Poiché gli attributi non vengono più indicati esplicitamente nelle azioni dello schema, non è più necessario distinguere le differenti occorrenze dello stesso simbolo.

Schema di traduzione modificato

$$
\begin{array}{lcll}
E & \to & E + T & \{p_2 \leftarrow \texttt{pop}().p;\ \texttt{pop}();\ p_1 \leftarrow \texttt{pop}().p; \\
 & & & \quad s \leftarrow \mathit{GOTO}(\texttt{top}().s, E); \\
 & & & \quad \texttt{push}(\langle s, \texttt{new node}('+', p_1, p_2)\rangle)\} \\
E & \to & T & \{p \leftarrow \texttt{pop}().p;\ s \leftarrow \mathit{GOTO}(\texttt{top}().s, E); \\
 & & & \quad \texttt{push}(\langle s, p\rangle)\} \\
T & \to & T \times F & \{p_2 \leftarrow \texttt{pop}().p;\ \texttt{pop}();\ p_1 \leftarrow \texttt{pop}().p; \\
 & & & \quad s \leftarrow \mathit{GOTO}(\texttt{top}().s, T); \\
 & & & \quad \texttt{push}(\langle s, \texttt{new node}('\times', p_1, p_2)\rangle)\} \\
T & \to & F & \{p \leftarrow \texttt{pop}().p;\ s \leftarrow \mathit{GOTO}(\texttt{top}().s, T); \\
 & & & \quad \texttt{push}(\langle s, p\rangle)\} \\
F & \to & (E) & \{\texttt{pop}();\ p \leftarrow \texttt{pop}().p;\ \texttt{pop}(); \\
 & & & \quad s \leftarrow \mathit{GOTO}(\texttt{top}().s, F);\ \texttt{push}(\langle s, p\rangle)\} \\
F & \to & \textbf{id} & \{p \leftarrow \texttt{pop}().p;\ s \leftarrow \mathit{GOTO}(\texttt{top}().s, F); \\
 & & & \quad \texttt{push}(\langle s, p\rangle)
\end{array}
$$

Parsing LL

- Passiamo ora ad analizzare il secondo caso di calcolabiltà efficiente degli schemi di traduzione, e precisamente quello relativo a schemi L-attributed unitamente a grammatiche di tipo LL.
- Anche in questo caso, l'esempio che ci guiderà sarà quello relativo alla grammatica per le espressioni modificata con l'eliminazione della ricorsione a sinistra.
- Abbiamo già visto come lo schema sia L-attributed.
- Abbiamo anche visto (astrattamente) come sia possibile eseguire correttamente lo schema nel caso si costruisca prima il parse tree, cioè eseguendo un opportuno attraversamento dell'albero stesso.
- Vogliamo ora in concreto vedere come modificare la procedura generale di parsing *LL* vista a suo tempo allo scopo di produre in output l'abstract syntax tree.

Organizzazione concreta del calcolo

- Nel caso di implementazione del parser LL mediante procedure ricorsive, un'implementazione dello schema di traduzione che rispetti le dipendenze funzionali è alquanto semplice (come in realtà già osservato).
- Sia $A \to X_1 \ldots X_{i-1} X_i \ldots X_k$ una generica produzione usata dal parser.
- Nel momento in cui, all'interno della procedura (per) A, viene chiamata la procedura X_i le procedure $X_1, \ldots, X_{i-1}$ sono già state chiamate e dunque eventuali valori da esse prodotti possono essere passati in input a X_i.
- Ciò consente di implementare il flusso dati richiesto dalla definizione di attributo ereditato, nel senso che, qualora X_i abbia attributi ereditati, questi dipendono da valori associati a (e calcolati dalle procedure) A e/o $X_1, \ldots, X_{i-1}$.

Organizzazione concreta del calcolo (2)

- In realtà, è proprio la struttura delle procedure ricorsive di un parser LL ad aver suggerito la definizione di attributo ereditato.
- Tornando alla nostra grammatica per le espressioni, consideriamo, ad esempio, la procedura per E (unica produzione $E \to TE'$) e le corrispondenti azioni dello schema:

$$E'.inh \leftarrow T.node \quad \text{e} \quad E.node \leftarrow E'.node.$$

- La procedura invocherà dapprima la routine T (che non necessita di "informazione ereditata").
- Il valore restituito da quest'ultima ($T.node$) sarà passato in input a E' come valore ereditato.
- Il valore restituito dalla chiamata ad E' sarà poi anche il valore restituito da E.

Organizzazione concreta del calcolo (3)

- Si noti quindi che le procedure corrispondenti ai non terminali sono più utilmente realizzate mediante funzioni.
- La funzione per E può quindi essere sinteticamente espressa nel modo seguente:

1: $T_{node} \leftarrow T()$
2: return $E'(T_{node})$

- Consideriamo, come altro esempio, la routine per E' (produzioni $E' \rightarrow +TE'$ e $E' \rightarrow \epsilon$).
- Abbiamo appena visto che essa riceve in input un argomento (attributo ereditato) che supporremo venga assegnato al parametro formale *inh*.

Organizzazione concreta del calcolo (4)

- Il corpo della funzione può essere espresso nel modo seguente:

```
1: if t ← get_next_token() =′ +′ then
2:     T_node ← T()
3:     inh_1 ← new node(′+′, inh, T_node)
4:     return E′(inh_1)
5: else if t =′ )′ or t =′ $′ then
6:     return inh
7: else
8:     error()
```

- Nel caso di implementazione ricorsiva del parser, non è dunque necessario che venga preliminarmente costruito il parse tree.